# Shawn Ciaurro

8/17/25

https://youtu.be/zwotQK5nkgY

# CS 405 Project Two Script

| **Slide Number** | **Narrative** |
| --- | --- |
| **1** | Intro: The following presentation will outline the principles and practices that Green Pace will adhere to in order to create a secure product during development. We will walk through threats, principals, policies, best practices and helpful tools that will help maintain our goal of secure coding and product development. |
| **2** | Multi Level security: Defense in depth is a concept of multiple layers of security to act as a failsafe in the event that one layer fails. [Defense in depth explanation through castle example]  Standardized principles: The policy is created to create a set of guidelines that developers can use to keep code and systems secure. Keeping with the standard principles keeps everyone on the same page when it comes to security and keeps security as a built-in focus rather than an optional component. Keeping with the guidelines will keep the production consistent and lead to fewer gaps in security.  Use of tools and tests: Use of automated scanners, code analysis tools, and unit test to reveal problems to be fixed before the software is released.  Universal rules: The rules laid out in the security policy should be applies to all projects by any individual with a hand in the project. |
| **3** | High Likely/ Med Threat: We’ll start by going over some of the moderately severe threats like weak error handling or weak privilege restrictions. They are common to come across and can be exploited but are less detrimental to security than the high threat rules.  High Likely/High Threat: These rules include problems such as lack of input validation, not regarding warnings and bad memory handling. They are common mistakes that are likely to occur and can cause catastrophic security breaches like buffer overflow attacks or injection attacks.  Low Likely/High Threat: The rules in this category are just as severely catastrophic for security as the previous category, however, they are less likely to occur because other safeguards in standard practices make them more difficult to exploit. |
| **4** | Here the rules are mapped to the outlined standards to demonstrate that they aren’t random rules but specifically align to our standards. Quickly going over each line we see:   1. **Validate Input Data**    1. STD-001 (Data types)    2. STD-002 (Check values before use)    3. STD-003 (Proper string handling)    4. STD-006 (Assertions)    5. STD-004 (Parameterized queries) 2. **Heed Compiler Warnings**    1. STD-001 (Data types) 3. **Architect and Design for Security Policies**    1. STD-005 (Memory handling)    2. STD-007 (Exception handling)    3. STD-009 (System calls)    4. STD-010 (Sensitive data clearing) 4. **Keep It Simple**    1. STD-002 (Check values)    2. STD-009 (Limit system calls) 5. **Default Deny**    1. STD-008 (Close files/resources) 6. **Principle of Least Privilege**    1. STD-005 (Memory handling)    2. STD-008 (Close files/resources)    3. STD-010 (Sensitive data clearing) 7. **Sanitize Data Sent to Other Systems**    1. STD-004 (Parameterized queries)    2. STD-009 (Limit system calls) 8. **Practice Defense in Depth**    1. STD-003 (String handling)    2. STD-010 (Sensitive data clearing) 9. **Use Effective Quality Assurance Techniques**    1. STD-006 (Assertions)    2. STD-007 (Exception handling) 10. **Adopt a Secure Coding Standard**     1. Applies to **all 10 standards** |
| **5** | * **STD-009 — Limit Use of System Calls and Shell Execution** (Level **5**; High/Likely) - This is at the top because of the shell/command injection risk. * **STD-003 — Proper String Handling to Avoid Buffer Overflows** (Level **5**; High/Likely) - Another small mistake that can lead to highly critical breaches. * **STD-004 — Parameterized Queries to Prevent SQL Injection** (Level **5**; High/Likely) - Like STD 9, it is important to protect from injection risks and parameterizing queries is an easy implementation that can save a system from high levels of risk. * **STD-002 — Check Data Values Before Use** (Level **4**; High/Likely)  Values must be validating and sanitized before being processed in the system to prevent logical failures, improper memory access, division by zero, and other vulnerabilities. * **STD-010 — Clear Out Sensitive Data After Use** (Level **4**; High/Unlikely) - Sensitive data like passwords or security keys may remain in memory after use, leaving them vulnerable to unauthorized access. Clearing or overwriting these values will make for more secure handling of the sensitive data. * **STD-005 — Prevent Memory Leaks & Handle Freed Memory** (Level **3**; Medium/Likely) - To properly manage memory, it must be ensured that freed memory is handled in a way to prevent reuse to avoid exploits of dangling pointers. * **STD-006 — Enforce Preconditions & Invariants with Assertions** (Level **3**; Medium/Likely) - Use of assertions helps identify logic errors by enforcing assumptions about the program’s state. This helps signal bugs before they can become vulnerabilities. It is important to note that this does not replace input validation but helps along side it. * **STD-007 — Use Targeted Exception Catching** (Level **3**; Medium/Likely) Proper exception handling is important to preserve the control flow and maintain stability. General exception catching using catch(…)does not reveal the root cause of the error. Better diagnostics can be achieved by using specific exception types. * **STD-008 — Close Files & Free Resources** (Level **3**; Medium/Likely) -  Memory leaks can be caused by the failure to close files. Other issues can also occur such as data corruption and file locks. Proper cleanup ensures the availability of resources for future uses. * **STD-001 — Use Appropriate Data Types for Variable Intent** (Level **2**; High/Unlikely) Choosing the correct data type for a variable’s application is essential to prevent behavioral and functionality issues, data loss, and memory access errors. The wrong data type selection opens the system up to multiple vulnerabilities, a few including overflow/underflow vulnerabilities, truncations, and memory access vulnerabilities. |
| **6** | Encryption at rest refers to securing stored data whether it be in databases or on hard drives. This is important because a bad actor can gain access to a database or a physical means of storage but still not be able to access the sensitive data without the decryption key. AES-256 encryption will be the Green Pace standard for encryption for all data stored at rest.  In flight: This type of encryption applies to data in transit from one system to another. Protecting data in transit by TLS 1.2 or higher will protect inter-system communications from attacks such as eavesdropping and man in the middle attacks. Places this will be necessary include login forms, third-party integrations and transmissions of sensitive data.  In use encryption deals with data that is in use in RAM. Policy should be that data is immediately cleared after use to prevent exposure of data through process inspection. |
| **7** | Authentication is used to verify the identity of a party attempting to gain access to a system. Policy for Green Pace systems will be multi-factor authentication for user logins and an additional layer of authentication for admin roles to access sensitive parts of the system. Logging all login attempts is used to audit activity.  Authorization is the concept of levels of user access for controls and authentication walls in a system. Role based access controls will be the Green Pace policy paired with the principle of least privileges for each role. Access to administrative functions, files and databases (CRUD functions) will be reserved for roles that require access with changes in user roles and new user addition going through an approval process.  Accounting is the process of tracking activity and access to audit and inspect to ensure compliance to security standards. This process is important to check that adherence to the standards is kept and to review actions to improve security in updates. |
| **8** | [Read from the slide] |
| **9** | [Read from the slide] |
| **10** | Does splitLoot divide evenly for valid inputs? **What it checks:** Positive path sanity check. **Result (show):** PASS — 10/2 = 5 returned. |
| **11** | Does splitLoot reject zero pirates? **What it checks:** Prevents divide-by-zero (vulnerability). **Result (show):** PASS — threw std::invalid\_argument. |
| **12** | Does splitLoot reject negative pirates? **What it checks:** Prevents invalid state (negative actors). **Result (show):** PASS — threw std::invalid\_argument. |
| **13** | Does splitLoot handle non-even splits without error? **What it checks:** Safe integer division; no crash on remainder. **Result (show):** PASS — 11/3 returned 3 (remainder ignored by policy) |
| **14** | Does splitLoot divide evenly for valid inputs? **What it checks:** Positive path sanity check. **Result (show):** PASS — 10/2 = 5 returned. |
| **15** | Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy.  Using the automated tools such as SonarQube, Cppcheck, Clang-Tidy, Fortify SCA, etc., makes it so that issues are caught early and throughout all of the stages of the development lifecycle. Catching possible risky or unsafe code early reduces the chances of the errors being built upon or making their way into the final product. During development of a project, automated plugins and IDE warnings provide instant feedback on developing code when then can be static tested with cppcheck or similar automated tests. Through the pre-production phases, Valgrind, Fortify SCA and similar tools can detect vulnerabilities by simulating runtime conditions. On the production side, security continues with tools that check that tamper-free code is being run safely. Integrity checkers and policy enforcement tools automatically check that the running system matches the predefined configurations. When tools detect problems, the system needs to respond and adapt to the issue with tools like WAF shields and automatic rollbacks triggered by dangerous events.  When tools detect problems, the system must respond and adapt quickly, and regular scans feed back into planning, so new threats inform updates to the security policy. This continuous loop makes security part of the pipeline rather than an afterthought. |
| **16** | The DevSecOps Pipeline integrates security considerations into every part of the SDLC.  The planning phase sees threat modeling and reviewing the defined secure coding policies.  During the coding phase, IDE plugins are used, built in prompts errors and warnings, and static testing.  During the build phase, the compiler generates warning and SonarQube SAST scans can be used.  In testing, unit test verify the proper functionality and dynamic analysis tools can be used.  At release container and configuration scans verify that no critical vulnerabilities are present before deployment.  Through the operation of the system, monitoring and response to attacks is used. |
| **17** | Some problems that will arise paired with their solutions are as follows:  Input validation gaps such as SQL injections or buffer overflows will be covered by validation/sanitation enforcement plus safer APIs.  Memory leaks or resource mismanagement can be caught by automated tools and static testing like Cppcheck, SonarQube and Valgrind.  To handle a problem of data left in memory we will clear sensitive data and adopt a least privilege design.  And finally, over reliance on manual reviews is solved with the employment of automation. |
| **18** | Waiting to act can cause damage that can be detrimental to the company’s security, finances, and trust. The longer the wait, the more chance for a breach and late fixes can cause further damages that will be more costly to fix than if they were addressed immediately. Acting now lessens these repercussions and is the obvious choice for our security needs. |
| **19** | Our security policy covers many vulnerabilities, however, there area still some gaps. Runtime protections remain limited, not all tools are integrated across every environment, and human error continues to pose risks without stricter enforcement.  Equifax breach of 2017: An unpatched Apache Struts vulnerability allowed during a runtime intrusion. Detection of this could have caught the attacker reducing their dwell time.  Capital One breach of 2019, hackers exploited a misconfiguration of roles showing how a slight human error can cause a catastrophic security failure.  To address these issues, we should expand runtime monitoring and incident response, increase automation coverage in our CI/CD pipeline, and conduct regular access reviews paired with ongoing threat modeling. Finally, developer training in secure coding will ensure that policies are consistently applied in practice. |
| **20** | In conclusion, the Green Pace security policy is made to integrate security into every part of the development process. The use of automation, tools and strict adherence to the policy will make the systems that Green Pace produces stronger, more secure, more economical and more trustworthy to the consumer. By following the steps in this presentation, the policy should be easily applied and ready to adopt. |
|  | [References] |